Printed by Jiffy Print Limited, Luton, Beds.

HiSoft DevpacST

Assembler/Debugger

Atari ST Computers

HISOFT

High Quality
Microcomputer
Software

DEVPACST

Welcome to DevpacST
DevpacST is a suite of programs designed to help you write and debug 68000 assembler programs: GenST is a macro assembler, with integrated screen editor, and MonST is a front panel disassembler and debugger.

The software described in this manual will definitely run on a 520ST with RAM loaded version of TOS and GEM, and a high resolution monitor is recommended for best use. It should work on a 260ST with ROM versions of the operating system, but we have as yet been unable to test this.

We strongly encourage you to make one back-up copy of DevpacST before using it.

If you cannot get DevpacST to run correctly on your computer then please contact your supplier for help or contact HiSoft directly:

HiSoft
180 High St North
Dunstable LU6 1AT
(0582) 696421

(C) Copyright HiSoft 1985

All rights reserved worldwide. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, without the written permission of the copyright holder. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature.

It is an infringement of the copyright pertaining to HiSoft DevpacST and associated documentation to copy, by any means whatsoever, any part of HiSoft DevpacST for any reason other than for the purposes of making a security back-up copy of the object code.

Motorola is a trademark of Motorola Inc.
WordStar is a trademark of MicroPro International.

Important: Subdirectories

Owing to a fatal bug in some versions of TOS, sub-directories are not supported when saving and loading text files from EdST when running on version 0.13. To see which version you have, included on the disc is a program called VERSION.PRG, which displays the version number. The bug is known to be fixed in version 0.19, but we do not know of any other versions, so if you have another you should test out sub-directories on a simple file before trusting anything important to them.

EdST deliberately makes no attempt to use subdirectories when running on version 0.13, so as a result attempting to load a file from one other than the default will give TOS error 33 (file not found), while a Save will save it in the current directory. The 'solution' is to copy GENST.PRG and GENST.RSC into each subdirectory that contains source files you wish to assemble.

The file README.S contains the latest information about your version of DevpacST, and other useful details which may be of use to you.

GenST - Macro Assembler and Full Screen Editor

GenST is a 68000 Macro Assembler integrated with a full screen editor, running under GEM. It enables you to type programs into your machine, to inspect and edit them at will, save them to disc, and assemble them, though not necessarily in that order. To begin with, the features of the screen editor will be covered, before the assembler itself.

The supplied disc contains at least four files:

GENST.PRG	-	the screen editor and assembler
MONST.PRG	-	the monitor/debugger/disassembler
GENST.RSC	-	a resource file for GENST, which must always be on the same disc and in the same directory as GENST.PRG
DEMO.S	-	the source to a simple demonstration program

EdST - Full Screen Editor

To load the program, double click on the application icon GENST.PRG from the desktop, as described in the ST User Guide. It will clear the screen, draw a window, and a menu bar, and is ready for you to type.
The cursor is denoted by an underline character, displayed as a '_'. The right hand side of the window shows a vertical scroll bar, the lower part a horizontal scroll bar, while at the top of the window there is a status line, showing various details about the text you are editing. To begin with, it should appear like this:

```
Line:    1  Col:     1  Free memory: 59980
```

(The memory figure may differ on your particular machine). What this means is that you are at the top left position of the file, and you have just under 60000 bytes free for text. The right hand side of the status area is used for displaying error messages, which disappear once any key is pressed.

The display in EdST is best looked upon as a window onto the current text. This window will be able to move both up and down, so you can view any part of the file, though currently no horizontal movement is allowed. If the window moves downwards then the text appears to move upwards, and this is called upward scrolling.

To enter text, simply type at the window, ending each line by pressing the Return key. Mistakes can be corrected using the Backspace key, which deletes the character to the left of the cursor, while the character above the cursor can be removed by pressing the Delete key. If you get stuck, pressing the Help key shows a summary of the commands not available from the menus. There are four sorts of ways to access the commands:

Using a single key, such as Return, Help, or a cursor key;
Clicking on a menu item in the usual way;
Using a menu shortcut, by pressing the Alternate key and another key simultaneously;
Using the Ctrl key in conjunction with another.

The Alt commands have been chosen to be easy and obvious to remember, while the Ctrl commands are based on those used in WordStar.

A Word About Dialog Boxes

EdST makes extensive use of dialog boxes, so it is worth recapping how to use them, particularly for entering text. EdST's dialog boxes contain buttons, radio buttons, and editable text.

Buttons can be clicked on with the mouse, and cause the dialog box to go away. Usually there is a default dialog box, shown by having a wider border than the others. Pressing Return on the keyboard is equivalent to clicking on the default button.

Radio buttons are groups of buttons that can only have one selected at a time - clicking on one automatically de-selects all the others.

Editable text is shown with a dotted line, and a vertical bar marks the cursor position. Characters can be typed into it, corrected using the Backspace, Delete and cursor keys, and the whole text can be removed by pressing the Esc key. If there is more than one editable text field in a dialog box, you can go between them by using the cursor up and down keys, or by clicking near them with the mouse.
Some dialog boxes only allow a limited range of characters to be typed into them - for example, the Goto Line dialog box only allows numeric digits to be entered.

EdST Commands

Moving the Cursor

The cursor can be moved around the screen by use of the cursor keys, to the right of the main keyboard, and via the scroll bars around the window. EdST prevents you going past the left- and right-hand edges of the window. If you cursor-up from the top line, the screen will scroll down a line, unless you are at the top of the file, when a message to that effect will appear in the status window. Similarly, cursor-down from the bottom displayed line will scroll the screen up, so long as you are not on the last line of the file. The arrows at each end of the scroll bars can also be used to move the cursor a character at a time.

For faster movement, the page up command is Ctrl R, and page down is Ctrl C. Alternatively, clicking in the grey area above the vertical slider will do a Page Up, and clicking in the lower grey area will do a Page Down.

To go to the top line in the file, choose Goto Top from the Options menu, or press the Alternate key (hence referred to just as Alt) and the T key. To move the cursor to the bottom of the file, choose Goto Bottom from the Options menu, or press Alt B.

To move quickly to another position in the current screen, simply click the mouse on the required character, and the cursor will move there.

Goto a particular line

To go to a particular line, choose Goto Line from the Options menu, or press Alt G, and a dialog box will appear. You can then type a number and press Return (equivalent to clicking in the OK button), or click in the Cancel button if you change your mind and do not wish to change your position in the file. So long as you do not Cancel, the window will display the required line (or the last line if too large a number was specified).

Searching for text

To find a particular section of text, choose Find from the Search menu (or press Alt F). A dialog box will appear, allowing you to enter your search string. If you click the Cancel button, no action will be taken; if you click Next (or press Return) the search will go forwards from the cursor position; clicking Previous makes the search go backwards. If the search was successful, the screen will be re-drawn at that point with the cursor positioned at the start of the string. If the string could not be found, the message 'Not found' will appear in the status window and the cursor remain unmoved. To find the next occurrence of the string, choose Next Forward from the Search menu (or press Alt N); to search for the previous occurrence, choose Previous from the Search menu (or press Alt P). The search starts at the position just past the cursor (for forward searches), or just before the cursor (for backward searches) and is case independent.

Search and Replace

This is not yet implemented in EdST, so all references to it on the Search menu and in the dialog box are dimmed.

Deleting text

Deleting lines

To delete the current line, press Ctrl Y. If this deletes the last line of the text, the window will be re-drawn showing just the new last line. To delete from the cursor position to the end of the current line, press Ctrl Q.

Clearing all the text

If you want to clear all the text currently being edited, choose Clear from the File menu (or press Alt C). If you have made any changes that have not been saved, a confirmation is required. You will return to the desktop.

Tabs

Tabs can be entered in the text by pressing the Tab key, and are stored as such. By default, the width of a tab is 7 spaces, but this can be changed to suit, by choosing Set Tabs from the Options menu (or pressing Ctrl T). You will be presented with a dialog box, allowing you to enter the new tab setting, which may be from 2 to 16 inclusive. When changing the tab setting, the whole window will be re-drawn to show the change. The assembler treats tabs in the same way as spaces, so you can enter your source code with tabs before instructions and between operators and comments, if you wish, for neater listings.

File Saving and Loading

To save your text file at any time, choose Save As from the File menu (or press Alt S); the standard GEM dialog box will appear, allowing you to select the disc and filename as required. Having made a selection, clicking the mouse in the OK button (or pressing Return) will save the file, removing any previous file with the same name, else click the mouse in the Cancel button to abort the command.

To load some text, choose Load from the File menu (or press Alt L). If you have made any changes to your file that have not been saved, a dialog box will appear warning you, with the option to Cancel the load, and preserve your text. If you do choose to load new text, the standard GEM dialog box will appear, allowing you to select the disc and filename. If there is an error during the load, an error number will appear in a dialog box. TOS error numbers are detailed in Appendix A.

It is usual for assembler source files to end with .S as an extension, so the file load and save dialog box only shows those files with that extension. If you wish to change the extension, press cursor-up to go to the directory name, delete the .S and replace with your choice (e.g. .ASM or .*), then click in the filename window to show the new list of files.

Inserting Files

If you wish to insert a file at the current cursor position, press Alt I, and you will be able to choose the name of the file to insert, memory permitting.

Command Summary

ALT Commands

A	Assemble
B	goto Bottom of file
C	Clear old text
F	Find string
G	Goto line
H	Help
I	Insert file at cursor position
L	Load text
N	find Next string
P	find Previous string
Q	Quit to desktop
S	Save text
T	goto Top of file

Ctrl keys

^C	page down
^Q	delete to end of line
^R	page up
^T	set tab distance
^Y	delete line

Function Keys

F1	block start
F2	block end
F3	save block
F4	copy block to cursor position
Shift F3	delete block

GenST 68000 Macro Assembler

In conjunction with EdST, GenST divides memory into two sections - a text buffer, and a symbol table. Normally these are 60000 and 50000 bytes respectively, but these figures can be changed by using the supplied Installation program. The text buffer is used by EdST for storing the text you enter, and the unused section is used by GenST for storing macro definitions. The symbol table is used for storing all labels required while assembling.

GenST is a two pass assembler; during the first pass it scans all the lines in memory and from disc if required building up a symbol table, while during the second pass the instructions are converted into bytes, a listing produced if required, and a .PRG file may be produced containing the assembled program. During the first pass, the only error messages produced will be fatal ones, which terminate the assembly. During the second pass errors and warnings will be shown, on the screen or to the printer, as well as a full listing and symbol table if required.

The assembler is invoked from within EdST by the Assemble item in the Options menu, or with the Alt A command; You will be presented with a dialog box allowing entry of the binary filename, and the method of listing you require. You can enter the name of the file in which the assembler will produce the object code, if required. If you don't want a binary file, just leave that area of the dialog box empty. For an executable program, the name should end with .PRG (for GEM programs), .TOS (for GEMDOS) or .TTP. (for GEMDOS programs that require parameters). There are also three radio buttons in the dialog box, to determine where any assembly listing should be directed. The None button disables any listing, the Screen button sends it to the EdST window, and the Printer button sends it to whatever the printer device is (set by the Install Printer desk accessory). As they are radio buttons, clicking on any one will de-select the others, as only one can be chosen. Clicking on Cancel will abort the assembly, while clicking on OK (or pressing Return) will start it. If you specified a binary file, an attempt will be made at this time to create it. If there is a problem, such as the disc being write protected, a TOS Error alert box will appear, showing the error, and you will be given the opportunity of changing the filename, or any other corrective action (e.g. changing the write-protect tab position).

Once a valid binary filename has been given (or if none has been specified) the window will clear, and assembly begin.

Firstly the message 'Pass 1' will be printed, and so long as there are no fatal errors then 'Pass 2' will follow it. During Pass 2 any listing requested will be produced, and screen output can be paused at any time by holding down the Alt key. Releasing the Alt key permits the listing to continue. If no listing was specified, the only lines shown will be those producing errors or warnings. When pass 2 has finished the messages 'xx errors found'

then 'xx lines assembled' will appear, followed by the amount of bytes used and available in the Symbol and Macro tables. The size of the Macro table is simply the amount of memory free in the text area, less 4 bytes. If a listing was specified this will be followed by a symbol table listing. The format of the instructions and directives accepted by the assembler will now be described.

When you assemble something subsequently, the dialog box will contain the options you previously set.

Assembler Statement Format

Each statement that is to be processed by GenST should have the following format:

```
LABEL     MNEMONIC  OPERANDS        COMMENT
Start     move.l    d0,(a0)+        store the result
```

Exceptions to this are comment lines, which are lines starting with an asterisk in column 1, and blank lines. Each field should be separated from the others by 'white space' - this is any number or mixture of space and tab characters.

Label field

The label should normally start at column 1, but if a label is required to start at another position then it should be followed immediately by a colon (':'). Labels are allowed on all instructions, but are prohibited on some assembler directives. A label may start with characters A-Z, a-z, _ (underline) or . (period), and may continue with a similar set, with the addition of the digits 0-9. Lower-case symbols are treated as the same as their upper case equivalents, and the first 16 characters are significant. Labels should not be the same as register names, or the reserved words SR, CCR, or USP - the assembler will not check this when examining the label field, but will get confused when they occur as an operand.

Mnemonic Field

The mnemonic field comes after the label field, and can be instructions, assembler directives, and macro calls. Optionally some instructions and directives allow a size specifier, separated from the mnemonic by a dot. Allowed sizes are .B byte, each particular case depend on the particular instruction or directive.

Operand Field

For those instructions or directives which require operands, this contains one or more operands, separated by commas.

Comment Field

Any white space (not within quotation marks) in an operand denotes the end of the operand, and anything following is treated as a comment and ignored by the assembler.

Addressing Modes

The available addressing modes are shown in the table below - note that GenST is not case sensitive.

Dn	data register direct, eg D3
An	address register direct, eg A5
(An)	address register indirect, eg (A1)
(An)+	address register indirect with post-increment, eg (a5)+
-(An)	address register indirect with pre-decrement, eg -(A0)
d(An)	address register indriect with displacement, eg 20(a7)
d(An,Rn.s)	address register indirect with index, eg 4(A6,D4.L)
d.W	absolute short address eg $3400.W
d.L	absolute long address, eg $12000.L
d(PC)	program counter relative with offset, eg value(PC)
d(PC,Rn.s)	program counter with index, eg value(PC,A2.W)
#d	immediate data, eg #26

Special addressing modes

CCR	condition codes register
SR	status register
USP	user stack pointer

In addition to the above, SP can be used in place of A7 in any addressing mode, eg 4(sp,d3.w)

Assembler Directives

Certain pseudo-mnemonics are recognised by GenST. These assemble directives, as they are called, are not decoded into opcode (with one exception), they simply direct the assembler to tak certain actions at assembly time. These actions have the effec of changing, in some way, the object code produced by GenST Directives are assembled exactly like executable instructions some may be preceeded by a label (for some it is obligatory) an may be followed by a comment. For compatibility, all directive can be preceeded by a period ('.'). The directives available are

<label> **END**
This directive signals that no more text is to be examined o this pass. It should always be the last non blank line of th text in memory, but is not obligatory to have an END statement.

<label> **EQU** <expression>
This must be preceded by a label, and sets the value of a label to the value of the expression. If there is any error in the expression (eg an undefined label), the value will not be assigned. The expression cannot include forward references.

<label> **EQUR** <register>
This must be preceded by a label, and allows registers to be denoted by a user name. An EQUR can only refer to data or address registers.

<label> **SET** <expression>
This must be preceeded by a label, and sets the value of a label to the value of an expression. It differs from EQU in that the label can be re-assigned to another, different value later on in the program. Forward refernces should not be used. At the start of pass 2 all labels defined on pass 1 with SET are made undefined.

<label> **DC.B** <operand(s)>
<label> **DC.W** <operand(s)>
<label> **DC.L** <operand(s)>
This directive is to define constants in memory. It may have one or more than one operand, multiple operands being separated by commas. The constants will be aligned on a word boundary if .W or single DC directive.

<label> **DS.B** <expression>
<label> **DS.W** <expression>
<label> **DS.L** <expression>
This directive is used to reserve memory locations, and the contents of the memory are not initialised in any way. The label will be set to the start of the area defined, which will be on a word boundary for DS.W and DS.L directives.

LIST
This directive turns the listing during Pass 2 on, and lines will be output to either the screen or printer, as specified at invocation time. All lines are listed until either an END or NOLIST directive is encountered. If there is no further NOLIST directive in the program, a symbol table will be listed at the end (unless directed otherwise with the OPT directive).

NOLIST
This directive turns any listing during Pass 2 **off**. If there is no further LIST directive in the program, no symbol table listing will be prodiced.

<label> **MACRO**
This directive is used to mark the start of a macro definition, and a label is obligitory. It is described in detail in the next section.

ENDM
This directive marks the end of the macro definition. It should not have a label.

IFEQ <expression>
If the expression evaluates to zero, the following instructions will be assembled, else they will be ignored, until an ENDC directive is encountered.

IFNE <expression>
If the expression evaluates to non-zero, the following instructions will be assembled, else they will be ignored, until an ENDC directive is encountered.

IFD <label>
If the label is defined, the following instructions will be assembled, else they will be ignored, until an ENDC directive is encountered. This should be used with care otherwise different object code could be generated on pass 1 and pass 2, producing meaningless code. It should really only be used in conjunction with the SET directive to conditionally define labels.

IFND <label>
If the label is not defined, the followinmg instructions will be assembled, else they will be ignored, until an ENDC directive is encountered. The warnings above under IFD apply equally to this directive.

ENDC
This terminates the current level of conditional assembly. It should NOT have a label.

INCLUDE <filename>
This powerful assembler command causes source code to be taken from a file from disc and assembled exactly as though it were explicitly present in the file. The directive must be followed by

a filename (separated from it by white space) - the filename should not be enclosed in quotes, and should follow normal TOS filename rules. In particular, it must not contain any spaces or tabs. A drive specified and extension may be included as required, e.g. INCLUDE b:myequs.asm
Include directives may be nested up to 8 levels.

<label> **OPT** <option(s)>
This controls various options in GenST, by a single alphabetic character, followed by a + for On, or a - for Off. Multiple options can be separated by commas. Valid options are*

M+ - macro expansions on	
M- - macro expansions off	(default)
S+ - symbol table listing on	(set by LIST)
S- - symbol table lsiting off	(set by NOLIST)
W+ - warnings on	(default)
W- - warnings off	

SPC <expression>
This outputs the number of blank lines in the expression in the assembly listing. The directive itself will not appear in the listing, and it has no effect if the listing is disabled (either by the starting options or a NOLIST directive).

LLEN <expression>
This sets the width of the assembly listing to be the given value. It has to be between 38 and 255, and defaults to 132.

PLEN <expression>
This sets the page length of the assembly listing, and is only valid when output is being directed to the printer. It defaults to 60.

TTL <string>
This sets the title that appears in assembly listings directed to the printer to be the given string.

Macro Operations and Conditional Assembly

GenST fully supports Motorola-style Macros, and limited conditional assembly.

Macro Operations

In GenST macros are a powerful tool that lets you greatly simplify assembly language programming. A macro may be defined thus:

```
CALL_TOS  MACRO
          MOVE.W #\1,-(SP)     put parameter on stack
          TRAP #1              call TOS
          LEA \2(A7),A7        restore stack
          ENDM                 end of macro
```

The directive MACRO is used to introduce a macro, and causes the label (which must preceed it) to be entered as a macro name. The label thus becomes the name by which the macro will be called. The macro definition ending is denoted with the ENDM directive. A typical macro call to the above macro could be:

```
MOVE.W #'X',-(SP)
CALL_TOS $02,4
```

When a macro is called, its name is used just like any other instruction, followed by up to 9 parameters, separated by commas. When the macro call is expanded by the assembler, it replaces any occurrences of \1, \2 etc with that particular parameter. In the above example, the code produced by the assembler would be

```
MOVE.W #'X',-(SP)
MOVE.W #$02,-(SP)
TRAP #1
LEA 4(A7),A7
```

In the listing, the default is to show just the macro call, and not the code produced by it. However, macro expansion listings can be switched on and off with the OPT directive.

If you wish a parameter to contain a space or a comma, you should enclose the whole parameter within < and > signs, eg

```
DO_PRINT <'DevpacST, the new standard'>,printer
```

As well as the parameters \1 to \9, there are two further macro parameters that can be used in definitions. \0 is used to denote the size specifier on the macro call, which defaults to 'W' if none is present, while \@ is useful for generating unique labels with each call of a macro. The \@ will be replaced with the sequence .nnn where 'nnn' is a count which increases by one with every macro call.

Macro calls may be nested up to a level of 8 deep, but macro definitions may not be nested.

The MEXIT directive stops prematurely the macro substitution, and if of limited use in this version of GenST, as it normally requires full condition assembly handling, which is not yet implemented.

Conditional Assembly

Conditional assembly allows you to write a program which has many internal options, chosen at assembly time. This can be very useful for generating, say, one program to run on two different machines, or to have some debugging routines included only in your internal versions of the software. There are five directives concerned with conditional assembly - the section of program to be conditionally assembled must start with an IFEQ, IFNE, IFD or IFND directive, and end with an ENDC directive. When using IFEQ, the section is assembled only if the expression is EQUAL to zero; when using IFNE, the section is assembled only if the expression is NOT EQUAL to zero; when using IFD, the section is asembled only if the label following it is defined; IFND is the opposite. Note that a label should NEVER be put on an IFNEQ, IFEQ, or ENDC directive, and the assembler will miss them, and generate confused code. As an example, if you were writing a program for the 260ST and the 520ST, and wanted a different amount of memory to be used depending on the machine, you could do something like this:

```
          .
m250ST    EQU 1               set to 1 for 260, 0 for 520
          .
          .
          .
          IFNE m250ST
          MOVE.L #1000,-(SP)
          ENDC
          IFEQ m250ST
          MOVE.L #2500,-(SP)
          ENDC
          .
          .
```

Current versions of GenST do not support nested conditional assembly.

Instruction Set

The complete 68000 instruction set is supported, and certain types are automatically accepted, so the line

```
     CMP.W Ending,A2
```

would be assembled as though it were

```
     CMPA.W Ending,A2
```

GenST makes automatic conversions to the A and I forms, though does not convert to the M or Q forms of some instructions. You can of course explicitly give the desired form of the A and I type instructions if you wish.

Position Independence

The ST is unique in current 68000-based machines in allowing position dependent code to run, using a special loader format. If you include any absolute addresses in your programs, GenST will automatically add the necessary relocation information on the end of the file in the standard GEMDOS format.
All code produced by GenST lies in the Text area of the file - the Data and Bss sections are of zero length, and are followed by any necessary relocation information.

Word Alignment

All instructions (with the exception of DC.B and DS.B) are always assembled on a word boundary. Should you require a DC.B explicitly on a word boundary, the directive EVEN can be used before it to force this.
Although all instructions that require it are word aligned, labels with nothing following them are NOT word aligned, and can thus have odd values. This is best illustrated by an example:

```
     NOP                      this will always be word aligned
     DC.B 'Odd'

START
     TST.L (A0)+
     BNE.S START
```

The above code would not produce the required result, as START would have an odd value. To help in finding such instructions, GenST will produce a warning if it finds an odd destination in a BSR or BRA operand. Note that nothing else checks it, so it is recommended that you preceed such labels with an EVEN directive if a DC.B (or DS.B) directive preceeds it.

Assembly Listing Format

Each line of the assembler listing generated during the second pass of GenST has the following format:

```
12044 00001860 3D7C40C00038          LABEL   RTS    all done
1     7        15                    35
```

The first entry in a line is the line number, which is a decimal integer from 1 to 65535. The line numbers correspond to lines in a particular file rather than lines in the assembly; thus after an INCLUDE directive the number restarts at 1, and when listing macro expansions the line number is not incremented. This makes it easy to move to the line of source code given in an error message or listing, either in the text you are currently editing or in an Included file.

The next entry, from column 7, is the value of the program counter, in hex.

The third entry, from column 15, is up to 20 characters (representing 10 bytes) in length and is the object code produced by the current instruction.

Columns 35 onwards contain the source code, as formatted when entered via EdST. Any tabs are expanded as per the current value of the tab setting in EdST.

Assembler Directive Summary

END	defines the end of the source code
EQU	assign value to a label
EQUR	assign a register equate to a label
SET	assign value to a label temporarily
DC	Define constant
DS	Define storage
LIST	turn listing on
NOLIST	turn listing off
SPC	print blank lines in listing
LLEN	set listing width
PLEN	set number of lines/page
TTL	set the heading title
IFEQ	assemble if equal to zero
IFNE	assemble if not equal to zero
IFD	assemble if label defined
IFND	assemble if label undefined
ENDC	end conditional assembly
MACRO	start of macro definition
ENDM	end of macro definition
MEXIT	exit the macro definition
INCLUDE	read source code from disc file
OPT	control various assembler options

Error Messages

When an error occurs, a message will be output of the form:
Error xx <message> at line yyyy in file z:vvv :
followed by a listing of the line. With fatal errors, a listing of the line will not be shown. Each error has a number, though this is largely unnecessary as each error is accompanied with an explanetory error message in English. Fatal error messages abort the assembly immediately, while Warnings are output to show something that, while not being an error, may be a mistake by the programmer.

Error numbers 1 through 50 are the normal types of errors, numbers 60 through 62 are warnings, and 80 through 84 are fatal.

1 - Unrecognised instruction
The instruction was not recognised - can also be caused by a call to an undefined macro.

2 - Undefined label
Reference was made to an undefined label

3 - Extra label in pass 2
This is an internal error and should never occur

4 - Label defined twice
The label at the start of this line has already been defined

5 - Phasing error
Currently this means the same as error 4, label defined twice

6 - Garbage following instruction
The instruction does not end with white space or an end of line

7 - Space expected
Two fields in a source line should have been separated by white space but wasn't

8 - Immediate data expected
This instruction requires immediate data

9 - Absolute expression required
This instruction or directive cannot have a relative expression

10 - Comma expected
Operands should be separated by a comma

11 - Data expected
An expression was expected in the line

12 - Data too large
The data is too large to fit in that size of instruction, also generated when branch instructions are out of range.

13 - Hex digit expected
A hex number should follow a $ sign

14 - Addressing mode expected
An addressing mode is missing

15 - Binary digit expected
A binary number should follow a % sign

16 -) expected
A close bracket was expected

17 - register expected
An address or data register was expected

18 - Unrecognised addressing mode
Usually means an addressing mode has been mistyped

19 - . expected
A dot was expected in the line

20 - W or L expected
Wrong size on instruction

21 - Addressing mode not allowed
That addressing mode is illegal for that instruction

22 - Byte sized not allowed
Illegal size for this instruction

23 - Address register expected

24 - String too long
String constants can be a maximum of 126 characters long

25 - No closing quote
A string ended without a closing quote

26 - .S or .L only
Illegal size for branch instruction

27 - Label expected on EQU
EQU must be preceeded by a label

28 - Not implemented

29 - Unbalanced ENDM
An ENDM has been found other than in a macro definition

30 - Macro nested too depply
Macros can only be nested to eight levels

31 - Unbalanced ENDC
An ENDC has been found when there is no previous IF

32 - Cannot nest IFs
This restriction will be removed

33 - Data register expected

34 - .W only
Word sized only allowed on this instruction

35 - Invalid expression
The expression does not make sense

36 - malformed register list
The register list in a MOVEM instruction is invalid eg D4-D1

37 - Long not allowed

38 - Include nested too deeply
Includes can only be nested to eight levels

39 - Binary file i/o error
An error occurred during writing the binary file (eg disc full)

40 - Absolute not allowed
This instruction or directive requires a relative expression

41 - bad register number

42 - illegal use of Macro label
Macro labels are not valid in expressions

43 - Size not allowed

44 - Size unrecognised
Valid sizes are .S, .W, .L and .B

45 - Unbalanced MEXIT
A MEXIT was encountered when there was no conditional assembly

46 - Unrecognised option
The option list was malformed

47 - Divide by zero error

48 - bad mixture of types
An operation was attempted on an illegal combination of operands
e.g. Relative+Relative

49 - forward reference not allowed
Forward references are not allowed in EQU or EQUR statements.

50 - relative not allowed
This instruction or directive does not permit a relative expression

Warnings

60 - Short branch of 0
This is illegal, so gets replaced with a NOP

61 - Sign extended operand
With MOVEQ the operand has to be sign extended to fit into 8 bits

62 - Jump to odd address
The destination of a Branch or BSR is odd

Fatal Errors

80 - expression MUST evaluate
Any error in an IF or DS expression is fatal

81 - Symbol table full
The symbol table is full

82 - File I/O failed
An error occurred during a read from an Include file - the TOS error number is also shown

83 - Macro used before defined
Currently this is not checked for - if you do this, GenST will get very confused

84 - Macro table full
There is no room to store any more macro definitions - check you haven't got a missing ENDM. The macro table is whatever space there is in the text area, so try and reduce the amount of source in memory by using INCLUDE.

During assembly the process can be aborted by pressing the ESC key during Pass 1 or Pass 2 - while not listing the key is checked less frequently, and may require a couple of quick presses before GenST notices. When pressed, the message 'Aborted' will appear, and all open files (ie any Included files and the binary file) will be closed.

MONST - DISASSEMBLER and DEBUGGER

MonST provides you with many facilities, including a full disassembler, to help you debug and test your machine code programs. It can debug both TOS and GEM programs and it saves and restores the screen display of the program you are debugging.

Getting Started

Your supplied DevpacST disc contains several files, the one we are interested in is called MONST.PRG, and to activate it simply double-click on its icon. It should not require installing, but if so then it should be configured as a GEM program, not a TOS program.

When it has loaded, the screen will clear and show four areas, which are special MonST windows. The lowest window is always used for keyboard input, while the others are used for various purposes. MonST uses its own screen drivers, instead of the standard MonST ones, for two reasons: firstly, it means that MonST writing to the screen will in no way effect your own programs writing to the screen, and secondly, they are extremely fast.
After loading, there will be a initial message in the third window, and a cursor in the lower one, inviting you to enter a program name. You should enter the filename of the program to be debugged, with any necessary extension, then an option if required. For non-GEM (i.e. TOS programs) you should follow the filename with a space, then the character T. This informs MonST that you require a clear VT52-type screen display, and not the graphic display the GEM programs require. If there is some problem in loading the specified file, the error number will be shown, and you can re-enter the filename. If you do not wish to debug a program, but just to look around in memory, or simply get used to MonST, entering Return on its own will do this.
On successfully loading a program, the message

MONST Debugger from HiSoft Author A.Pennell
Breakpoint

will appear (if you pressed Return as a command line the message will be similar except that the message 'Address Error' replaces 'Breakpoint').

MonST is around 12k in length, but requires around 32k of memory for its workspace, mostly required to keep a copy of the screen display of the program you are debugging.
To remove the entry message, press any key, and the windows will clear and be replaced with a 'front panel' display of four sections:

The top window displays the 68000 register values, and the memory they address in hex, including the Program Counter (PC), the Supervisor Stack Pointer (A7'), and the status register (SR). The

meaning of the flags of the SR is also shown, with the addition of T for Trace, U for User mode or S for Supervisor.

The next section displays 64 bytes around the Memory Pointer (shown by '>') in hex, separated into Word boundaries. If any section of memory does not physically exist, it is displayed as asterisks.

The third window shows a disassembly of three instructions, starting at the one at the Memory Pointer. On entry, the value of the Memory Pointer will be the first instruction in the program being debugged, or an undefined location if no filename was given.

The four, lowest window is where all keyboard interaction and commands take place.

Commands are entered from the keyboard in response to the cursor and may be entered in either upper or lower case. Certain commands whose effects are potentially disastrous require the pressing of the CTRL key as well as the command letter. Throughout the MonST manual the use of the CTRL key is denoted by the symbol '^' e.g. ^Z means hold the CTRL and Z keys simultaneously.

Commands take effect immediately - there is no need to terminate them with RETURN, and invalid commands are simply ignored. The relevant sections of the 'front panel' are updated after every command is processed so that you can observe any results of the particular command.

Many commands require the input of a hex number, which can be preceeded by a minus sign. If by force of habit you preceed your hex numbers with a '$' sign, they will still be accepted. Numbers greater than $FFFFFFFF will be taken MOD $100000000, and may be truncated to the 68000 addressing range and also forced to be even, depending on the function. Pressing Return on its own returns a value of 0, and pressing ESC will abort the command.

After entering MonST, you can return to the Desktop by pressing ^C, though this is an 'emergency exit' and should be used only when your program under inspection is not capable of returning to the DeskTop in the usual way.

THE COMMANDS AVAILABLE

MonST is a multiple exception handler, and thus runs in Supervisor mode. It can be entered by an Address Error, Illegal Instruction, Privilege Violation, or Divide by Zero. It uses the illegal instruction $4AFA as a breakpoint, but the Trap instructions are left alone, allowing thier use by your own programs.

The following commands are available from within MonST. In this section, whenever Return is used to terminate an input this in fact can be any non-alphanumeric character, with the exception of ESC - this will abort any command if pressed.

ENTER **increment the memory pointer by word**
increment the memory pointer by a word (or a byte if its current value is odd).

'-' (minus) **decrement memory pointer by a word**
decrement the memory pointer by a word (or 3 bytes if its currently odd).

'G' **search memory ('G'et a sequence)**
You are asked 'Search for B/W/L/T/I?', standing for Bytes, Words, Long words, Text, or Instructions. For B/W/L, you will be prompted to enter subsequent parameters (followed by Return) until you have defined the whole sequence, then press Return on its own to start the search, beginning from the current memory pointer value+1. If you select Text, you will be prompted for a string of characters ending in Return. The Instruction option is a very powerful command, and searches for sections of instructions. For example, entering the string '$14(A' would find instructions such as MOVE.L D2,$14(A0) (the case of the characters is not important, and it goes without saying that this is very much slower than the other search options).

'N' **find Next occurrence**
This can be used after the G command to find subsequent occurrences of the data. With B/W/L/T options, you will always find at least one occurrence, which will be in the buffer within MonST that the sequence is stored in. The T option may also find an occurrence in the keyboard buffer. With those options mentioned, the ESC key is tested during the search every 64k bytes, to interrupt it. With the I option selected, it is tested every 2 bytes.

'I' **Intelligent copy**
This is used to copy a block of memory from one location to another - it is intelligent in that the block of memory may be copied to locations where it would overlap its previous locations. 'I' prompts for the inclusive start and end addresses of the block to be copied ('First?', 'Last?') and then for the address to which the block is to be copied ('To?'); enter suitable numbers in response to each prompt. If the start address

is greater than the end address then the command is aborted, otherwise the block is moved as directed.

'M' **set the Memory Pointer to a specified address**
You are prompted to enter an address, then the memory pointer and the front panel is updated.
'M' is essential as a prelude to entering code, tabulating memory etc.

'P' **Print a disassembly, optionally to the printer**
You are first prompted to enter the 'Start' and 'End' addresses of the code you wish to disassemble, and if the start address is greater than the end address then the command is aborted. The next prompt is 'Printer Y/N?' to select printer or screen output. Once the disassembly has begun it may be aborted by pressing ESC.

'Q' **Quick disassembly**
This gives a screenful of instructions starting from the memory pointer, and pauses at the end of each screenful. Any key except ESC will continue the disassembly.

'R' **Register modify**
This allows the alteration of the displayed register values. The prompt 'Register' will appear, and wait for one of 'D' (data register), 'A' (address register), 'P' (program counter) or 'S' (status register). For data and address registers, a number from 0 to 7 must also be pressed. You will then be prompted for a value for the register, then the front panel updated. (Note that the Supervisor Stack Pointer cannot be altered by this command is it is required for MonST).

'T' **Temporarily update memory pointer**
This allows you to temporarily inspect areas of memory pointed to by a register. You will be prompted for a register name (as above without SR), and the front panel will be updated with its value as the memory pointer. The question 'Update Y/N?' will appear, and if you press Y the update to the memory pointer will be permanent, else it will return to its previous value.

'V' **View other screen**
This shows the screen display of the program being debugged; pressing any key returns you to the front panel.

'W' **fill memory With pattern**
This prompts for 'First', 'Last' and 'With', for the inclusive area to be filled, then the byte with which it will be filled.

'X' **eXamine block of memory**
This displays a block of memory starting from the Memory Pointer in screenfulls at a time, showing 16 bytes per row in hex, then ASCII (or as '.' if not a valid printable character). After each page, pressing ESC will return you to the front panel, else any other key will display the next pageful.

'^B' set/reset Breakpoint at memory pointer

Breakpoints are defined as being the illegal instruction $4AFA, and there can be up to 8 simultaneous breakpoints at any one time. This command places a breakpoint at the Memory Pointer location, while keeping an internal copy of what it replaced. When it is executed, MonST will be entered with a suitable message, and the original instruction replaced. This command can generate three possible errors - 'Its odd!', 'Its ROM!', or 'Too many!' - all of these abort the command. If this is done on an existing breakpoint, then it is cleared. (Placing a breakpoint within MonST will crash it).

'^A' insert breakpoint after current instruction, then execute it

This inserts a breakpoint at the instruction following the memory pointer, then executes it. This can be particularly useful for Trap and BSR instructions if you do not wish to single step through the relevant subroutine, but just see what happens on its return. This can generate the same errors as the '^B' command above.

'^C' quit from MonST

This returns from MonST directly to the desktop, and should be used with caution. Ideally it should only be used when MonST is invoked with an empty command string, but can be used as an 'Emergency exit' from debugging an application if required.

'^K' Kill all breakpoints

All breakpoints are removed, by replacing the $4AFAs with their original contents.

'^R' Return from exception

This will perform an RTE from MonST back to whatever was the last thing to call it via any exception. It will go back to the displayed value of the PC, with the register values shown. The Trace bit of the SR will be reset, but it will be otherwise unaltered. If you wish to jump to a certain address instead, alter PC (with the R command) then ^R to jump to it.

'^X' eXamine memory to printer

This is similar to the X command, except that output is directed to the printer device. It can be stopped by pressing Esc.

'^Z' single step an instruction

The Trace bit of the SR is enabled, the instruction at the Memory Pointer is executed, and the front panel re-entered ready for the next instruction. This can single step any software, including ROM, but there are some hardware I/O instructions that should not be single stepped.

Modifying memory

To enter data into memory, set the Memory Pointer to the desired location (using Modify), then bytes, words or lomg words can be entered in hex. To do this, simply type the required number using hex digits only; if terminated with Return it will be taken to be a size comparable to its value. For example, entering the digits

1B4E [Return] will be taken as Word sized. To specify the size explicitly terminate the sequence with W or L to suit, and the data will be entered in that size.

Debugging Graphic Software

To aid in debugging graphic software, MonST keeps a copy of the current screen memory in its own buffer, saving it on entry and restoring it on exit. The only by-product is that MonST deliberately disables mouse movement while it is in control, to prevent it corrupting the screen display. Every time control passes to your application, the mouse is re-enabled.

When debugging TOS programs (specified with the T in the command line) the VT52 cursor is made to flash after its contents have been restored.

Command Summary

```
G     - Get a sequence (search)
I     - Intelligent copy
M     - Modify memory pointer
N     - find Next match (after G)
P     - Print disassembly to printer or screen
Q     - Quick disassembly to screen
R     - Register alter
T     - Temporarily update memory pointer
V     - View other screen
W     - fill memory With byte
X     - eXamine memory
^A    - put breakpoint after current instruction then execute it
^B    - set/reset breakpoint
^C    - quit to the desktop
^K    - Kill all breakpoints
^R    - Return to whatever called MonST
^X    - eXamine memory to printer
^Z    - single step

0-9, A-F  - enter data into memory
Return    - increase memory pointer by a word
-         - decrease memory pointer by a word
```


A Short Tutorial in the use of GenST and MonST

Included on the DevpacST disc is the source code to a simple GEMDOS program, to aid in your understanding of EdST, GenST and MonST. Follow the tutorial through exactly the first time, then you can try and 'do you own thing' afterwards, once you are more familiar with it. The program prints a simple message, waits for a key, the returns to the desktop. We shall assemble, run, then debug this program.

Assembling the demo

Double-click GenST, and wait until the window and cursor appears. Now move the mouse and select the Load option from the File menu, and a dialog box will appear, showing all the files on that disc, ending with .S, the standard filetpe for assembly source code. One of the files will be DEMO.S, so move the mouse over the name itself, and double-click on it. The disc drive should whir into action, and after a short wait the screen should clear and the program be displayed. It is slightly larger than a screenful, so press Ctrl C to page down, and you should see the rest of the file.

With most programs its best to have a trial assembly, that doesn't produce a listing or binary file, to check the syntax of the source. Although DEMO.S has no errors, we'll do it anyway. Move the mouse to the Options menu, and click on Assemble.

A dialog box will appear, showing a binary file of nothing, which is what we require, and the None box selected, which we also require. Click in the Assemble box, and assembly will begin.

After a short while, assembly should finish, with no errors, and you should press any key to return to the editor. Now that we are satisfied there are no syntactic errors, we can assemble it properly, so again choose Assemble from the Options menu.

When the dialog box appears, type DEMO.TOS, to specify the binary file. Clicking assemble will then start assembly once more, though it will be slightly slower as it has to create the binary file on the disc. When it finishes, press any key to return to the editor, then choose Quit from the File menu. If you inadvertantly made any changes, an alert box will appear, and you should click OK on it to ignore the changes you made. You will then return to the desktop.

To test the program, double click on the DEMO.TOS icon, and it should work as expected. Press any key and you will return to the Desktop.

Although it works, we shall now use MonST to trace through the program, step by step. Load MonST by double-clicking on it, then type DEMO.TOS T (there is a space before the T) and press Return. The program should now load, and the 'Breakpoint' message appear. Press any key and the front panel display should appear.

The disassembly window should show the PEA instruction, followed by the calling sequence to GEMDOS. Before executing this, press V, which shows you the screen of the program being debugged, and in our case this is an empty screen, with the flashing VT52 cursor. Pressing any key will return the front panel display.

Press ^Z to single step, and the memory and disassembly windows will be updated to reflect the new value of the program counter, and press ^Z again to execute the MOVE. If you look at the top window, and at the hex next to A7, you will see a word of 9 on the stack, which is what one would expect after executing that instruction. The next instruction is a TRAP #1, and its not very wise or informative to single-step the whole TOS routine for printing a string, so press ^A. This puts a breakpoint at the next instruction, and executes the Trap. You will then get a Breakpoint message, so press any key to continue. If you now press V so can see the result of the trap - the message.

Press ^Z twice more, until you reach the next Trap. This one waits for a key press, so press ^A, and the screen will display the programs screen, and wait for a keypress, so press the space bar. You will get another breakpoint message, so press another key, and the value in D0.B should be $20, the ASCII code for space. The final Trap returns control to TOS, so to let the program run its course press ^R.

Appendix A - TOS Error Numbers

-1	fundamental error
-2	drive not ready
-3	unknown command
-4	CRC error
-5	bad request
-6	seek error (disc not present)
-7	unknown media
-8	sector not found
-9	no paper
-10	write fault
-11	read fault
-12	general error
-13	write protect
-14	media change
-15	unknown device
-16	bad sectors on format
-17	insert other disc
-32	invalid function number
-33	file not found
-34	path not found
-35	too many open files
-36	access denied
-37	invalid handle
-39	out of memory
-40	invalid memory block address
-46	invalid drive specified
-49	no more files
-64	range error
-65	internal error
-66	invalid program load format
-67	setblock failure

TOS error numbers are normally negative, and MonST displays them as such. However, the GEM TOS error alert box used in GenST displays them as positive numbers.

Licence Agreement For DEVPAC ST

By opening the diskette envelope the user agrees to use the software product on a single designated microcomputer system only. The use of the software on any other microcomputer is not authorized.

The user agrees not to permit copies to be made for use by persons on any equipment other than the designated microcomputer system.

The user acknowledges that the unauthorized distribution or use of software received from HiSoft will cause material damage to HiSoft.

Please fill out the enclosed registration card, sign it and mail it back to HiSoft at the following address:

HiSoft

HIGH QUALITY MICROCOMPUTER SOFTWARE

180 High Street North, Dunstable, Beds LU6 1AT
Telephone: (0582) 696421

REGISTRATION CARD

I have read the licence agreement for

_____
(INSERT PRODUCT NAME)

and agree to all terms and conditions of the licence.
I hereby request to be registered as the authorized user of the software.

Serial #: _____ Version: _____

Name: _____
(PLEASE PRINT)

Company: _____

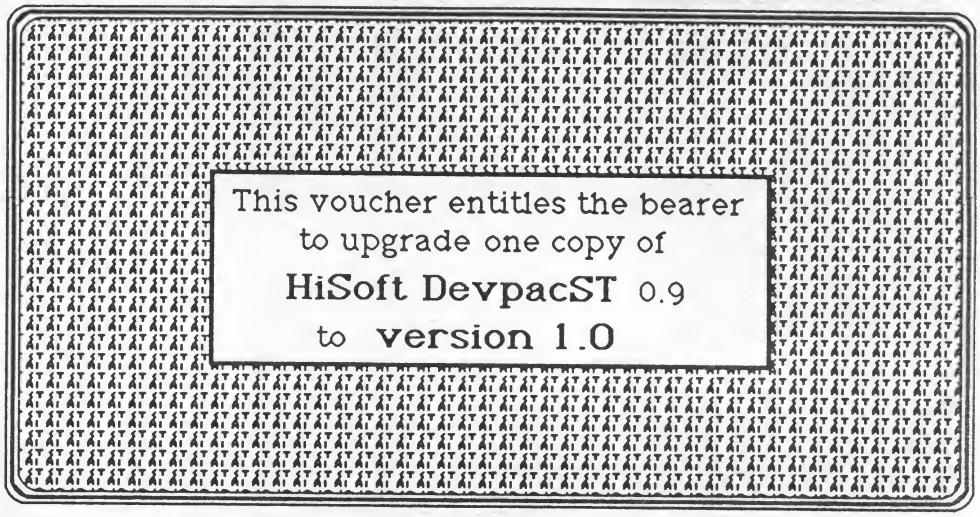
Street: _____

City: _____ County/State: _____

Postcode/Zip: _____

Signature: _____ Date: _____

Note: HiSoft will not provide customer service and updated versions of products to customers not registered with us.



Please
Affix
Stamp

HiSoft
180 High Street North
Dunstable
LU6 1AT
ENGLAND